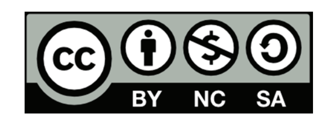


Tema 2: Interfaz Alto Nivel – Ensamblador

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya





The trouble with programmers is that you can never tell what a programmer is doing until it's too late.

- *Seymour Cray*

As soon as we started programming, we found out to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

- *Maurice Wilkes – 1949*

If debugging is the process of removing bugs, then programming must be the process of putting them in.

- *Edsger W. Dijkstra*

Índice

Referencias históricas

Evolución desde el punto de vista de la Arquitectura (Lenguaje Máquina):

- **1978.** Se anuncia el i8086 como una extensión del i8080. El i8086 es un microprocesador de 16 bits. Se queda a medio camino entre una máquina de acumulador y una máquina de registros de propósito general.
- **1980.** Se anuncia el i8087, coprocesador en coma flotante. El LM del i8086 se amplía con 60 instrucciones. Se abandona el concepto de acumulador y se utiliza un híbrido entre un banco de registros y una pila para las operaciones en coma flotante.
- **1982.** El i80286 aumenta el espacio de direcciones a 24 bits. Se ofrece el *real addressing mode* para seguir ejecutando aplicaciones i8086.
- **1985**. El i80386 extiende la arquitectura a 32 bits. Se añaden nuevos modos de direccionamiento e instrucciones. El nuevo procesador es casi una máquina de registros de propósito general.
- En los siguientes procesadores apenas hay cambios en la Arquitectura (4 nuevas instrucciones en 10 años).
- **1997**. El Pentium MMX incluye instrucciones para aplicaciones multimedia (MMX). El conjunto de instrucciones multimedia se ha ido aumentando en los siguientes procesadores: SSE (Pentium III, 1999), SSE2 (Pentium 4, 2001), SSE3 (Pentium 4 Prescott, 2005; ampliado AMD Athlon, 2005), SSE4 (Core y AMD K10, 2007).
- **2003**. AMD realiza la extensión de la arquitectura a 64 bits. Los registros aumentan a 64 bits y se aumenta su número a 16. Intel tuvo que copiar esta extensión.

Referencias históricas

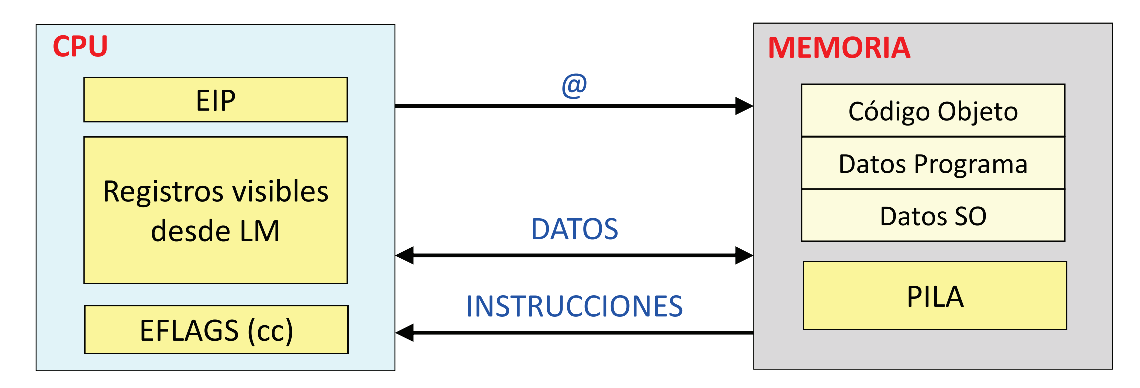
Lenguaje Máquina MIPS (características RISC)

- **Instrucciones aritméticas acceden sólo a registros**
  - En algunos casos un operando puede ser inmediato
- **Solo las instrucciones Load y Store pueden accede a memoria.**
- **Referencias a memoria con modos de direccionamiento simples**
  - **Base + Desplazamiento**
- **Instrucciones de longitud fija**
- **Muchos registros**

Lenguaje Máquina x86 (características CISC)

- **Instrucciones pueden referenciar diferentes tipos de operandos**
  - inmediato, registros, memoria
- **Instrucciones aritméticas pueden leer/escribir en memoria, pero sólo 1 de los 2 operandos puede estar en memoria**
- **Referencias a memoria pueden suponer cálculos complejos**
  - **Rb + S*Ri + D**
- **Instrucciones pueden tener diferente longitud**
- **Pocos registros**

Visión del programador



- **EIP: Contador de programa. Apunta a la siguiente instrucción a ejecutar**
- **Registros: Se usan muy frecuentemente como variables de acceso rápido**
- **Códigos de Condición**
  - Almacenan información respecto al comportamiento de las últimas instrucciones ejecutadas
  - Se usan en los saltos condicionales
- **Memoria**
  - Vector direccionable a nivel de byte, Little endian
  - Código, datos usuario, datos SO
  - Pila para soportar gestión de subrutinas

Visión del programador

- **EIP: Contador de programa. Apunta a la siguiente instrucción a ejecutar**
- **Registros: Se usan muy frecuentemente como variables de acceso rápido**
- **Códigos de Condición**
  - Almacenan información respecto al comportamiento de las últimas instrucciones ejecutadas
  - Se usan en los saltos condicionales
- **Memoria**
  - Vector direccionable a nivel de byte, Little endian
  - Código, datos usuario, datos SO
  - Pila para soportar gestión de subrutinas

Características del ensamblador

- **Tipos de datos básicos**
  - Enteros
    - ✓ dato de 1, 2 ó 4 bytes
    - ✓ datos y direcciones (punteros)
  - Reales (coma flotante): 4, 8 ó 10 bytes
  - No incluye tipos estructurados
    - ✓ Se codifican como datos almacenados de forma contigua
- **Operaciones primitivas**
  - Operaciones aritméticas sobre registros y datos en memoria
  - Transferencia de datos entre memoria y banco de registros
  - Saltos condicionales e incondicionales (a/de procedimientos)

Visión del programador

¿Qué necesitamos estudiar?

- Espacio de memoria
- Registros disponibles
- Repertorio de instrucciones: qué hacen, cómo se codifican, cuánto tardan
- Tipos y representación de los datos
- Modos de direccionamiento
- Secuenciamiento de instrucciones
- Comunicación con el exterior

Visión del programador

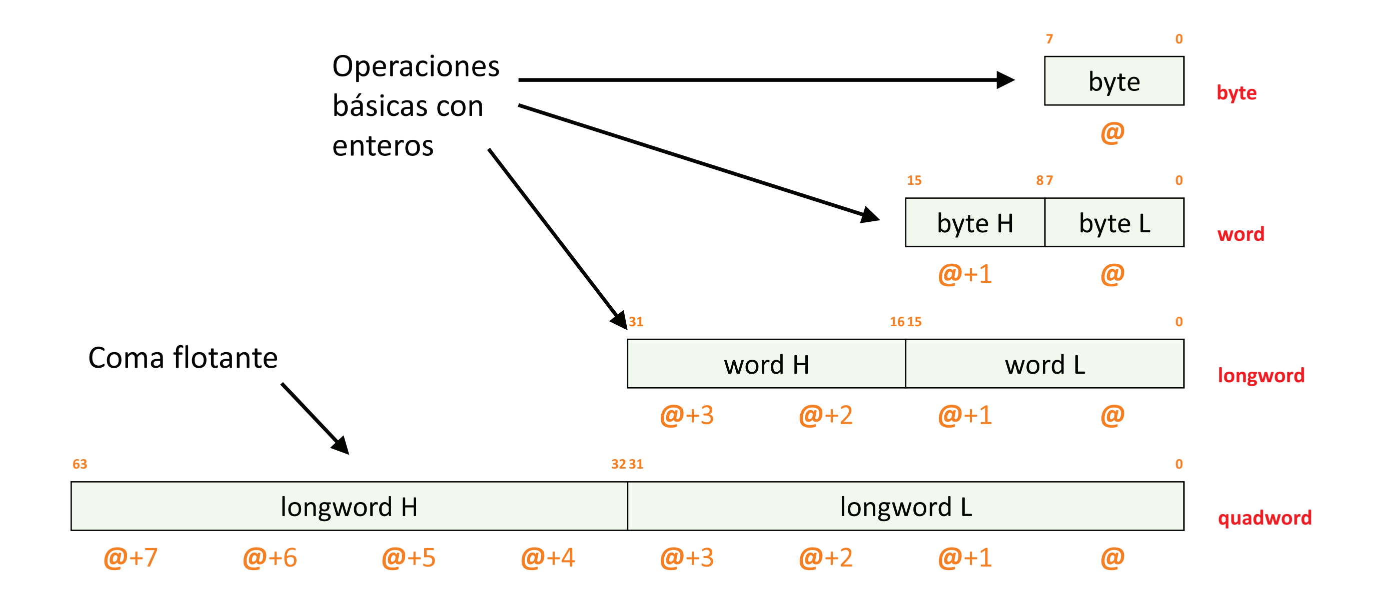
Espacio de memoria

- Espacio lineal de 2^{32} posiciones de 1 byte
- Modo protegido / Modelo plano de memoria/ Little endian

Registros disponibles

32 bits	16 bits	8 bits	
%eax	%ax	%ah, %al	AH AL / AX / EAX
%ebx	%bx	%bh, %bl	
%ecx	%cx	%ch, %cl	
%edx	%dx	%dh, %dl	
%esi	%si		SI / ESI
%edi	%di		
%esp	%sp		Reservados para uso específico de subrutinas
%ebp	%bp		
%eip			Contador programa
%eflags			Palabra de estado

Tipos de datos básicos



Tipos de datos básicos

0	34
1	22
2	5A
3	3B
4	C1
5	45
6	FF
7	00
8	21
9	2A
10	2C
11	7B
12	90
13	43
14	11
15	FF

```
byte 8:  0x21
byte 3:
word 8:
word 3:
longword 8 :
longword 3 :
quadword 8 :
quadword 3 :
```

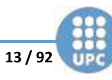
¡ Little Endian !

Tipos de datos básicos

0	34
1	22
2	5A
3	3B
4	C1
5	45
6	FF
7	00
8	21
9	2A
10	2C
11	7B
12	90
13	43
14	11
15	FF

```
byte 8:  0x21
byte 3:
word 8:
word 3:
longword 8 :
longword 3 :
quadword 8 :
quadword 3 :
```

¡ Big Endian !



Tipos de datos básicos

Rango Naturales

- **byte**: 0 ≤ x ≤ 255
- **word**: 0 ≤ x ≤ 65.535
- **longword**: 0 ≤ x ≤ 4.294.967.215

Rango Enteros

- **byte**: -128 ≤ x ≤ 127
- **word**: -32.768 ≤ x ≤ 32.767
- **longword**: -2.147.483.648 ≤ x ≤ 2.147.483.647

Rango Reales (IEEE 754)

- **Precisión simple** (32 bits, 24 precisión): $1{,}18 \cdot 10^{-38} \leq x \leq 3{,}40 \cdot 10^{38}$
- **Precisión doble** (64 bits, 53 precisión): $2{,}23 \cdot 10^{-308} \leq x \leq 1{,}79 \cdot 10^{308}$
- **Precisión doble extendida** (80 bits, 64 precisión): $3{,}37 \cdot 10^{-4932} \leq x \leq 1{,}18 \cdot 10^{4932}$

Modos de direccionamiento

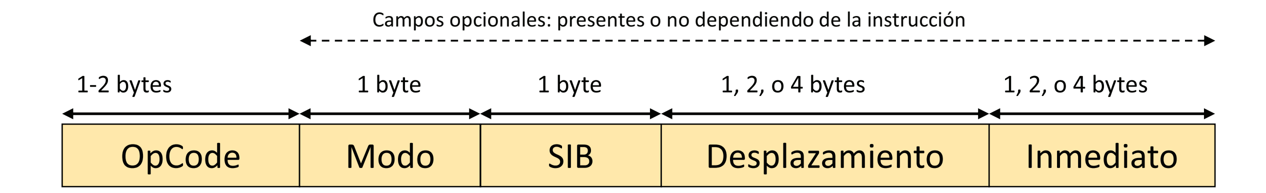
- **Inmediato**: **$19, $-3, $0x2A, $0x2A45**
  - Codificado con 1, 2 ó 4 bytes
- **Registro**: **%eax, %ah, %esi**
- **Memoria**: **D(Rb, Ri, s)** → **M[Rb+Ri×s+D]**
  - D: desplazamiento codificado con 1, 2 ó 4 bytes
  - Rb: registro base. Cualquiera de los 8 registros
  - Ri: registro índice. Cualquiera excepto %esp
  - S: factor escala: 1, 2, 4 u 8

Modos de direccionamiento

- **Ejemplos de modos de direccionamiento:**

```
(%eax,%ebx)          M[eax+ebx]
-3(%eax,%ebx)
(%eax,%ebx,4)
(,%ebx,4)
12(%eax)
(%eax)
3(%eax,%esi,2)
4
$4
%eax
%al
```

Codificación de las instrucciones

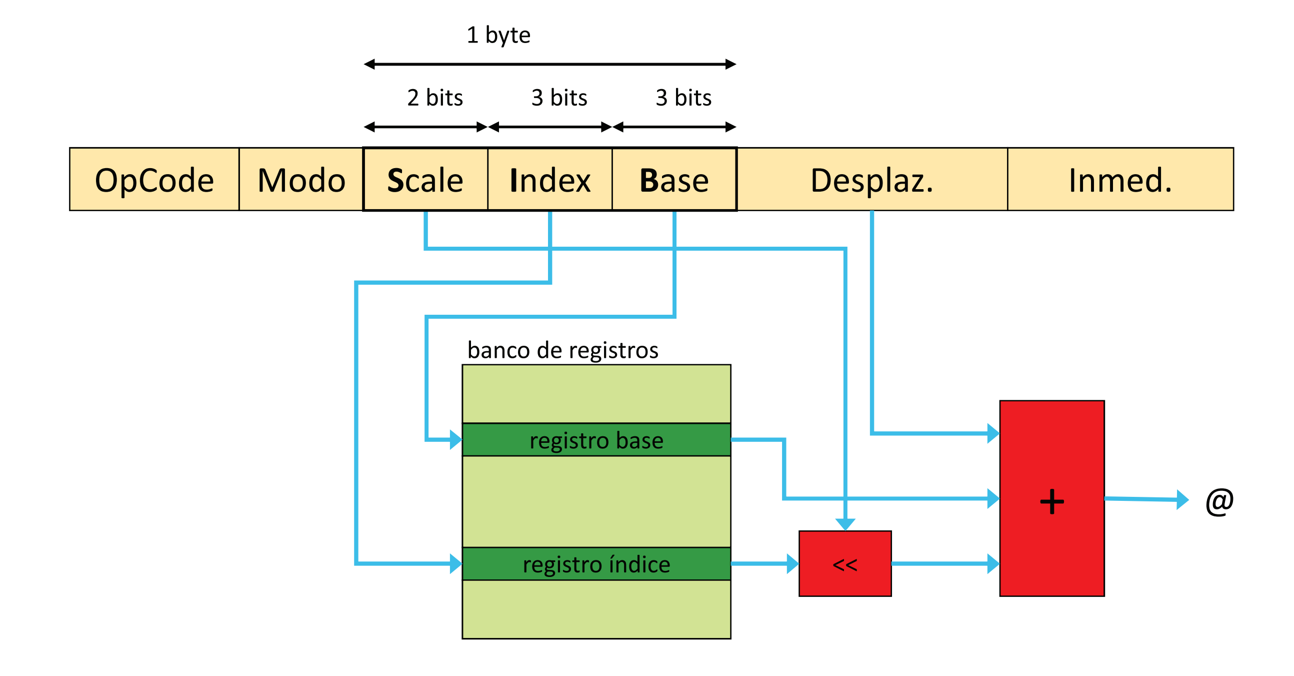


Formato General de las instrucciones

- **OpCode codifica:**
  - la operación a realizar
  - el tamaño de los operandos
  - cuál es el operando fuente y cuál el destino
  - si el operando fuente es un inmediato o registro/memoria
- **Modo codifica:**
  - el modo de direccionamiento del operando memoria si lo hay
  - el registro para los operandos registro
  - indica si hay desplazamiento para el caso de que un operando esté en memoria
- **SIB, en el caso que uno de los operandos esté en memoria, codifica:**
  - el escalado (**S**cale)
  - registro índice (**I**ndex)
  - registro base (**B**ase)



Codificación del modo memoria



Posibles valores de Scale: 0,1,2,3 (equivale a multiplicar por 1,2,4,8 respectivamente)

IA-32 Intel® Architecture Software Developer's Manual Volume 2: Instruction Set Reference

Instrucciones

- Movimiento de datos

Instrucciones	Descripción	Notas	Ejemplo
MOVx op1, op2	op2 ← op1	x = {L, W, B}	MOVB $-1,%AL
MOVSxy op1, op2	op2 ← ExtSign(op1)	xy = {BW, BL, WL}	MOVSBW %CH,%AX
MOVZxy op1, op2	op2 ← ExtZero(op1)	xy = {BW, BL, WL}	MOVZWL %BX,%EDX
PUSHL op1	%ESP ← %ESP – 4; M[%ESP] ← op1		PUSHL 12(%EBP)
POPL op1	op1 ← M[%ESP]; %ESP ← %ESP + 4;		POPL %EAX
LEAL op1, op2	op2 ← &op1	op1: memoria	LEAL (%EBX,%ECX),%EAX

Instrucciones

- Aritméticas

Instrucciones	Descripción	Notas	Ejemplo
ADDx op1, op2	op2 ← op2+op1	x = {L, W, B}	ADDL $13,%EAX
SUBx op1, op2	op2 ← op2-op1	x = {L, W, B}	SUBW %CX,%AX
ADCx op1, op2	op2 ← op2+op1+CF	x = {L, W, B}	ADCL %EDX,%EAX
SBBx op1, op2	op2 ← op2-op1-CF	x = {L, W, B}	SBBL %ECX,%EAX
INCx op1	op1 ← op1+1	x = {L, W, B}	INCL %EAX
DECx op1	op1 ← op1-1	x = {L, W, B}	DECW %BX
NEGx op1	op1 ← -op1	x = {L, W, B}	NEGL %EAX

Instrucciones

■ **Aritméticas (2)**

Instrucciones	Descripción	Notas	Ejemplo
IMUL op1, op2	op2 ← op2·op1	op2: registro	IMUL (%EBX),%EAX
IMUL inm,op1,op2	op2 ← op1·inm	inm: constante	IMUL $3,%EAX,%ECX
IMULL op1	%EDX%EAX ← op1·%EAX	op1: mem. o reg. (**Enteros**)	IMUL L (%EBX)
MULL op1	%EDX%EAX ← op1·%EAX	op1: mem. o reg. (**Naturales**)	MUL L (%EBX)
CLTD	%EDX%EAX ← ExtSign(%EAX)	x = {L, W, B}	INCL %EAX
IDIVL op1	%EAX ← %EDX%EAX / op1 %EDX ← %EDX%EAX % op1	op1: mem. o reg. (**Enteros**)	IDIVL (%EBX)
DIVL op1	%EAX ← %EDX%EAX / op1 %EDX ← %EDX%EAX % op1	op1: mem. o reg. (**Naturales**)	DIVL %ESI



Instrucciones

■ **Lógicas**

Instrucciones	Descripción	Notas	Ejemplo
ANDx op1, op2	op2 ← op2&op1	x = {L, W, B}	ANDL $13,%EAX
ORx op1, op2	op2 ← op2|op1	x = {L, W, B}	ORW %CX,%AX
XORx op1, op2	op2 ← op2^op1	x = {L, W, B}	XORL %EDX,%EAX
NOTx op1	op1 ← ~op1	x = {L, W, B}	NOTB %AH
SALx k,op1	op1 ← op1<<k (aritm.)	x = {L, W, B}, k: inm. o %CL	SALL #1,%EAX
SHLx k,op1	op1 ← op1<<k (log.)	x = {L, W, B}, k: inm. o %CL	SHLW #CL,%DX
SARx k,op1	op1 ← op1>>k (aritm.)	x = {L, W, B}, k: inm. o %CL	SARL #1,%EAX
SHRx k,op1	op1 ← op1>>k (log.)	x = {L, W, B}, k: inm. o %CL	SHRW #CL,%DX
CMPx op1, op2	op2-op1	x = {L, W, B}, activa flags	CMPL $13,%EAX
TESTx op1, op2	op2&op1	x = {L, W, B}, activa flags	TESTW %CX,%AX

Instrucciones

- **Secuenciamiento**

Instrucciones	Descripción	Notas	Ejemplo
JMP etiq	EIP ← EIP+despl.	EIP ← &etiq	JMP loop
JMP op	EIP ← op	op: reg. o mem.	JMP (%ebx,%esi,4)
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {E, NE, G, GE, L, LE, ...} (**Z**)	JLE else
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {A, AE, B, BE, ...} (**N**)	JA loop
Jcc etiq	if (cc) EIP ← EIP+despl.	cc = {Z, NZ, C, NC, O, ...} (flags)	JNC error
CALL etiq	%ESP ← %ESP-4 M[%ESP] ← EIP EIP ← EIP+despl.	Guardar @retorno EIP ← &etiq	CALL sub
CALL op	%ESP ← %ESP-4 M[%ESP] ← EIP EIP ← op	op: reg. o mem.	CALL (%EBX)
RET	EIP ← M[%ESP]; %ESP ← %ESP+4		RET



Códigos de condición (FLAGS)

- **Se activan implícitamente después de ejecutar cualquier instrucción aritmética**
- **Se almacenan en un registro (32 bits) especial del procesador: EFLAGS**

ADDL op1, op2 ; op2 ← op2 + op1

- **CF** (Carry Flag): Carry de la suma del bit 31. Overflow en unsigned
- **ZF** (Zero Flag): ZF = 1 si t == 0
- **SF** (Sign Flag): SF = 1 si t < 0
- **OF** (Overflow Flag): OF = 1 si (a>0 && b>0 && t<0) || (a<0 && b<0 && t>0)

CMPL op1, op2 ; op2 - op1, y se activan los flags sin guardar el resultado de la resta

- **CF** (Carry Flag): Carry (borrow) de la resta del bit más significativo
- **ZF** (Zero Flag): ZF = 1 si a == b
- **SF** (Sign Flag): SF = 1 si (a - b) < 0 (⇒ a<b)
- **OF** (Overflow Flag): OF = 1 si (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

Instrucciones

- **Secuenciamiento**

Instrucciones	Flags	Descripccion
JE etiq	ZF	Igual / cero
JNE etiq	~ZF	No igual / no cero
JS etiq	SF	Negativo
JNS etiq	~SF	No negativo
JG etiq	~(SF^OF)&~ZF	Mayor (con signo)
JGE etiq	~(SF^OF)	Mayor o igual (con signo)
JL etiq	(SF^OF)	Menor (con signo)
JLE etic	(SF^OF)|ZF	Menor o igual (con signo)
JA etiq	~CF&~ZF	Mayor (sin signo)
JAE etiq	~CF	Menor o igual (sin signo)
JB etiq	CF	Menor (sin signo)
JBE etiq	CF^ZF	Menor o igual (sin signo)

Códigos de condición (FLAGS)

- **Ejemplos**

Instrucciones	OF	DF	IF	TF	SF	ZF	AF	PF	CF
ADD op1, op2	X				X	X	X	X	X
AND op1, op2	0				X	X	?	X	0
DEC op1	X				X	X	X	X	
NOT op1									
STC									1
MOV op1, op2									
MUL op1	X				?	?	?	?	X
SAL k,op1	i				X	X	?	X	X
LEAL op1, op2									

Significado	
X	Depende resultado
0	cero
1	uno
?	No definido
	No modificado
i	Consultar manual

- **Para más detalles, ¡consultad el manual!**

Ejemplo ensamblador

- Convertir 'abc…' en 'ABC…'

```
.data
   .align 4
v: .string "Esto es una frase … salto de linea.\n"
.text
   .align 4
   .globl main
   .type main,@function
main: ...
      xorl %esi,%esi          ; esi ← 0
      movl $v, %ebx           ; ebx ← @inicio v
do:   movb (%ebx, %esi),%al   ; al ← v[i]
      cmpb $'a',%al           ;
      jl cont                 ; ¿v[i] < 'a'?
      cmpb $'z',%al
      jg cont                 ; ¿v[i] > 'z'?
      andb $0xDF,(%ebx,%esi)  ; v[i] ← MAY(v[i])
cont: incl %esi               ; i++
      cmpb $'\n', (%ebx,%esi) ;
      jne do                  ; ¿v[i] == '\n'?
end:  ...
```

```
addb $'A'-'a',(%ebx,%esi)
```

CÓDIGOS ASCII			
A	0100 0001	a	0110 0001
B	0100 0010	b	0110 0010
C	0100 0011	c	0110 00??
…		…	

Ejemplo ensamblador

- $A_i \leftarrow A_i / B_i$, A y B vectores de enteros acabados en 0

```
.data
   .align 4
A: .int 34, 45, 12, …, 56, -67, 0
B: .int -4, 6, 91, …, 12, 4, 0
.text
   .align 4
   .globl main
   .type main,@function
main: ...
      xorl %ecx,%ecx            ; ecx ← 0
      leal A, %edi              ; edi ← @inicio A
      movl $B, %esi             ; esi ← @inicio B
do:   movl (%edi,%ecx,4),%eax   ; eax ← A[i]
      cltd                      ; edx ← ExtSign(eax)
      idivl (%esi,%ecx,4)       ; eax ← edxeax/B[i]
      movl %eax,(%edi,%ecx,4)   ; A[i] ← eax
      incl %esi                 ; i++
      cmpl $0,(%esi,%ecx,4)     ;
      jne do                    ; ¿B[i] == 0?
end:  ...
```

Traducción de sentencias C a ensamblador

- Sentencia CONDICIONAL (IF-THEN-ELSE)

Modelo:

```
if (cond)
  CUERPO-IF
else
  CUERPO-ELSE
```

Traducción genérica:

```
        evaluar condición
        j(no cumple) else
if:     CUERPO-IF
        jmp endif
else:   CUERPO-ELSE
endif:
```

Ejemplo:

```
int max(int x, int y) {
  int max;
  if (x>y) max = x;
      else max = y;
  return max;
}
```


Traducción de sentencias C a ensamblador

- Sentencia CONDICIONAL (IF-THEN-ELSE)

Ejemplo:

```
int max(int x, int y) {
  int max;
  if (x>y) max = x;
      else max = y;
  return max;
}
```

Traducción:

```
max:    pushl %ebp
        movl %esp, %ebp
        movl 8(%ebp),%ecx
        movl 12(%ebp),%edx
        cmpl %edx,%ecx
        jle else
if:     movl %ecx,%eax
        jmp endif
else:   movl %edx,%eax
endif:  popl %ebp
        ret

# x → 8[%ebp]
# y → 12[%ebp]
# resultado en %eax
```

Traducción de sentencias C a ensamblador

- Sentencia ITERATIVA (DO-WHILE)

Modelo:

```
do
  CUERPO-DO
while (cond)
```

Traducción genérica:

```
do:   CUERPO-DO
      evaluar condición
      j(cumple) do
end:
```

Ejemplo:

```
int ContA(char v[]) {
  int i, cont;
  cont = 0;
  i = 0;
  do {
    if (v[i] == 'a') cont++;
    i++;
  } while (v[i] != '.');
  return cont;
}
```


Traducción de sentencias C a ensamblador

- Sentencia ITERATIVA (DO-WHILE)

Ejemplo:

```
int ContA(char v[]) {
  int i, cont;
  cont = 0;
  i = 0;
  do {
    if (v[i] == 'a') cont++;
    i++;
  } while (v[i] != '.');
  return cont;
}
```

Traducción:

```
ContA:  pushl %ebp
        movl %esp, %ebp
        movl 8(%ebp),%ecx
        xorl %eax,%eax
        xorl %edx,%edx
do:     cmpb $'a',(%ecx,%edx)
        jne endif
        incl %eax;
endif:  incl %edx
        cmpb $'.',(%ecx,%edx)
        jne do
end:    popl %ebp
        ret

# @v → 8[%ebp]
```

- Sentencia ITERATIVA (WHILE)

Modelo:

```
while (cond) {
  CUERPO-WHILE
}
```

Traducción genérica:

```
while: evaluar condición
       j(no cumple) end
       CUERPO-WHILE
       jmp while
end:
```

Ejemplo:

```
int gcd(int a, int b) {
  while (b!=0) {
    if (a>b) a = a-b;
         else b = b-a;
  }
  return a;
}
```


Traducción de sentencias C a ensamblador

- Sentencia ITERATIVA (WHILE)

Ejemplo:

```
int gcd(int a, int b) {
  while (b!=0) {
    if (a>b) a = a-b;
         else b = b-a;
  }
  return a;
}
```

Traducción:

```
gcd:    pushl %ebp
        movl %esp, %ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%ecx
while:  cmpl $0,%ecx
        je end
        cmpl %ecx,%eax
        jle else
        subl %ecx,%eax
        jmp endif
else:   subl %eax,%ecx
endif:  jmp while
end:    popl %ebp
        ret
# a → 8[%ebp]
# b → 12[%ebp]
```

Traducción de sentencias C a ensamblador

- Sentencia ITERATIVA (WHILE)

Ejemplo:

```
int gcd(int a, int b) {
   int t;
   while (b!=0) {
      t = b;
      b = a%b;
      a = t;
   }
   return a;
}
```

Traducción:

```
gcd:    pushl %ebp
        movl %esp, %ebp
        movl 8(%ebp),%eax
        movl 12(%ebp),%ecx
while:  cmpl $0,%ecx
        je end
        pushl %ecx
        cltd
        idivl %ecx
        movl %edx,%ecx
        popl %eax
        jmp while
end:    popl %ebp
        ret
# a → 8[%ebp]
# b → 12[%ebp]
```


Traducción de sentencias C a ensamblador

- Sentencia ITERATIVA (FOR)

Modelo:

```
   for(INI; COND; INC) {
      CUERPO-FOR
   }
```

Ejemplo:

```
int sumV(int V[], int N) {
   int sum, i;
   sum = 0;
   for (i=0; i<N; i++)
      sum = sum + V[i];
   return sum;
}
```

Traducción genérica:

```
      INI
for:  evaluar condición
      j(no cumple) end
      CUERPO-FOR
      INC
      jmp for
end:
```

Traducción de sentencias C a ensamblador

- Sentencia ITERATIVA (FOR)

Ejemplo:

```
int sumV(int V[], int N) {
  int sum, i;
  sum = 0;
  for (i=0; i<N; i++)
    sum = sum + V[i];
  return sum;
}
```

Traducción:

```
sumV: pushl %ebp
      movl %esp, %ebp
      movl 8(%ebp),%edx
      xorl %eax,%eax
      xorl %ecx,%ecx
for:  cmpl 12(%ebp),%ecx
      jge end
      addl (%edx,%ecx,4),%eax
      incl %ecx
      jmp for
end:  popl %ebp
      ret

# @V → 8[%ebp]
# N → 12[%ebp]
```


Traducción de sentencias C a ensamblador

- Sentencia CONDICIONAL (SWITCH)

Ejemplo:

```
switch_eg(int x)
{
  int result = x;
  switch (x) {
   case 100: result *= 13; break;
   case 102: result += 10;
   case 103: result += 11; break;
   case 104:
   case 106: result *=result; break;
   default: result = 0;
  }
  return result;
}
```

Implementación con una serie de condicionales (tipo if):
- Funciona bien en algunos casos
- Muy lento en la mayoría

Implementación con vector de punteros:
- Más eficiente en general

Traducción de sentencias C a ensamblador

- Sentencia CONDICIONAL (SWITCH)

Ejemplo:

```
void S(int x)
{
  int result = x;
  switch (x) {
   case 100: result *= 13; break;
   case 102: result += 10;
   case 103: result += 11; break;
   case 104:
   case 106: result *=result; break;
   default: result = 0;
  }
  return result;
}
```

Traducción con IFs:

```
S:    pushl %ebp
      movl %esp, %ebp
      movl 8(%ebp),%eax
      cmpl $100,%eax
      jne C102
L100: imull $13,%eax,%eax
      jmp end
C102: cmpl $102,%eax
      jne C103
L102: addl $10,%eax
L103: addl $11,%eax
      jmp end
C103: cmpl $103,%eax
      je L103
C104: cmpl $104,%eax
      je L106
C106: cmpl $106,%eax
      jne default
L106: imull %eax,%eax
      jmp end
def:  xorl %eax,%eax
end:  popl %ebp
      ret
```


Traducción de sentencias C a ensamblador

- Sentencia CONDICIONAL (SWITCH)

Ejemplo:

```
void S(int x)
{
  int result = x;
  switch (x) {
   case 100: result *=
   case 102: result +=
   case 103: result +=
   case 104:
   case 106: result *=
   default: result = 0
  }
  return result;
}
```

Traducción con Vector de punteros (pseudoC):

```
code *JT[7] = {L100,LDEF,L102,L103,L104,LDEF,L106};

xi = x - 100;
if ((x<100)||(x>106)) jmp LDEF;
goto JT[x-100];
L100: {código para x==100}; goto DONE;
L102: {código para x==102};
L103: {código para x==103}; goto DONE;
L104:
L106: {código para x==106}; goto DONE;
LDEF: {código para default};
DONE: return result;
```

Traducción de sentencias C a ensamblador

- Sentencia CONDICIONAL (SWITCH)

Ejemplo:

```
void S(int x)
{
  int result = x;
  switch (x) {
   case 100: result *= 13; break
   case 102: result += 10;
   case 103: result += 11; break
   case 104:
   case 106: result *=result; br
   default: result = 0;
  }
  return result;
}
```

Traducción con Vector de punteros :

```
.section .rodata
     .align 4
LT:  .long L0, LD, L2, L3, L46, LD, L46
S:    pushl %ebp
      movl %esp, %ebp
      movl 8(%ebp),%eax
      cmpl $100,%eax
      jl LD
      cmpl $106,%eax
      jg LD
      leal -100(%eax),%edx
      jmp LT(,%edx,4)
L0:   imull $13,%eax      # case 100
      jmp end
L2:   addl $10,%eax       # case 102
L3:   addl $11,%eax       # case 103
      jmp end
L46:  imull %edx,%edx     # case 104,106
      jmp end
LD:   xorl %eax,%eax      # default
end:  popl %ebp
      ret
```


Tipos de datos estructurados

Vectores

- **Declaración en C:**

  `tipo nombre[tamaño];   //indexado a partir de 0`

- **Almacenamiento en posiciones consecutivas de memoria**
  - Acceso elemento V[i]: **@inicio V + i·tam** (tam: tamaño de los elementos de V)

- **Ejemplos:**

Declaración en C	Tamaño elemento	Tamaño vector	@elemento i
`char A[12];`	**1B**	12B	@inicio A + i
`char *B[80];`	**4B**	320B	@inicio B + **4**·i
`double C[1024];`	**8B**	8KB	@inicio C + **8**·i
`int *D[5];`	**4B**	20B	@inicio D + **4**·i
`int E[100];`	**4B**	400B	@inicio E + **4**·i

Tipos de datos estructurados

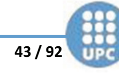
- **Vectores**

Ejemplo:

```
int Vi(int V[100], int i) {
   return V[i];
 }
```

Traducción:

```
Vi: pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp),%ecx     # @V → 8[%ebp]
    movl 12(%ebp),%edx    # i → 12[%ebp]
    movl (%ecx,%edx,4),%eax
    popl %ebp             # resultado en %eax
    ret
```

Tipos de datos estructurados

Matrices

- **Declaración en C:**
  `tipo nombre[NumFilas][NumColumnas]; //indexado a partir de (0,0)`
- **Almacenamiento por filas en posiciones consecutivas de memoria**
  - Acceso elemento A[i][j]: **@inicio A + (i·NumColumanas + j)·tam** (tam: tamaño de los elementos de V)
- **Ejemplos:**

Declaración en C	Tamaño elemento	Tamaño matriz	@elemento (i, j)
`char A[80][25];`	1B	2000B	@inicio A + i·25 + j
`char *B[80][10];`	4B	3200B	@inicio B + (i·10 + j) · 4
`double C[1024][100];`	8B	800KB	@inicio C + (i·100 + j) · 8
`int *D[5][90];`	4B	1800B	@inicio D + (i·90 + j) · 4
`int E[100][30];`	4B	12000B	@inicio E + (i·30 + j) · 4

Tipos de datos estructurados

Matrices 3-dimensiones

- **Ejemplo, matriz de enteros de 3 dimensiones:**
  `int M3D[10][64][48]; // cada int ocupa 4 bytes`

- **La matriz se almacena en posiciones consecutivas de memoria: cara a cara y en cada cara por filas.**
- **Acceso al elemento `M3D[cara][fila][columna]`:**
  - `@inicio M3D + (cara·64·48 + fila·48 + columna)·4`

- **Es fácil deducir como se almacenan / accede a matrices de N-dimensiones.**

Tipos de datos estructurados

- **Matrices**

Ejemplo:

```
int Mfc(int M[50][80], int fil, int col) {
   return M[fil][col];
 }
```

Traducción:

```
Mfc: pushl %ebp
     movl %esp, %ebp
     movl 8(%ebp),%ecx      # @M → 8[%ebp]
     movl 12(%ebp),%edx     # fil → 12[%ebp]
     imull $80,%edx,%eax
     addl 16(%ebp),%eax     # col → 16[%ebp]
     movl (%ecx,%eax,4),%eax
     popl %ebp              # resultado en %eax
     ret
```

Tipos de datos estructurados

- **Matrices**

Ejemplo:

```
void Copia(int M[50][80], int X[50][80]) {
    int i, j;
    for (i=0; i<50; i++)
      for (j=0; j<80; j++)
        M[i][j] = X[i][j];
 }
```


Tipos de datos estructurados

- **Matrices**

Traducción:

```
Copia: pushl %ebp             # i → %ecx
       movl %esp, %ebp        # j → %edx
       movl 8(%ebp),%edi      # @M → 8[%ebp]
       movl 12(%ebp),%esi     # @X → 12[%ebp]
       salvar reg
       xorl %ecx, %ecx
fori:  cmpl $50, %ecx
       jge endi
       cuerpo-FORi
       incl %ecx
       jmp fori
endi:  restaurar reg
       popl %ebp
       ret
```

```
#cuerpo-FORi:
       xorl %edx, %edx
forj:  cmpl $80, %edx
       jge endforj
       cuerpo-FORj
       incl %edx
       jmp forj
endj:
```

```
#cuerpo-FORj:
imull $80, %ecx, %eax
addl %edx, %eax
movl (%esi,%eax,4),%ebx
movl %ebx,(%edi,%eax,4)
```

Instrucciones ejecutadas: 13 + 50·(7 + 80·8) = 32.265

Tipos de datos estructurados

- Matrices

Objetivo: reducir el número de instrucciones.
• Se puede ver la matriz como un vector de 4000 posiciones.

Optimización:

```
Copia: pushl %ebp
       movl %esp, %ebp
       movl 8(%ebp),%edi   # @M → 8[%ebp]
       movl 12(%ebp),%esi  # @X → 12[%ebp]
       salvar reg
       xorl %ecx,%ecx
loop:  movl (%esi,%ecx,4),%eax
       movl %eax,(%edi,%ecx,4)
       incl %ecx
       cmpl $4000, %ecx
       jl loop
       restaurar reg
       popl %ebp
       ret
```

Instrucciones ejecutadas: 11 + 4000·5 = 20.011

Tipos de datos estructurados

- Matrices

Objetivo: reducir el número de instrucciones.
• El bucle se ejecuta 1/2 de veces.

Optimización: Desenrollar

```
Copia: pushl %ebp
       movl %esp, %ebp
       movl 8(%ebp),%edi   # @M → 8[%ebp]
       movl 12(%ebp),%esi  # @X → 12[%ebp]
       salvar reg
       xorl %ecx,%ecx
loop:  movl (%esi,%ecx,4),%eax
       movl %eax,(%edi,%ecx,4)
       movl 4(%esi,%ecx,4),%eax
       movl %eax,4(%edi,%ecx,4)
       addl $2,%ecx
       cmpl $4000,%ecx
       jl loop
       restaurar reg
       popl %ebp
       ret
```

Instrucciones ejecutadas: 11 + 2000·7 = 14.011

Tipos de datos estructurados

- **Instrucciones SIMD (Single Instruction Multiple Data)**

Optimización: Desenrollar 4 + SIMD

```
Copia: pushl %ebp
       movl %esp, %ebp
       movl 8(%ebp),%edi   # @M → 8[%ebp]
       movl 12(%ebp),%esi  # @X → 12[%ebp]
       salvar reg
       xorl %ecx,%ecx
loop:  movdqa (%esi,%ecx,4),%xmm0
       movdqa %xmm0,(%edi,%ecx,4)
       addl $4,%ecx
       cmpl $4000,%ecx
       jl loop
       restaurar reg
       popl %ebp
       ret
```

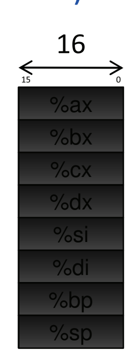
Objetivo: reducir el número de instrucciones.
- El bucle se ejecuta 1/4 de veces y además tiene menos instrucciones

`movdqa`: mov double quadword (128 bits) aligned
(dirección de inicio de X y M debe ser múltiplo de 16)
(existe `movdqu` u=unaligned pero es menos eficiente)
`%xmm0`: registro de 128 bits para las extensiones SSE
(en 128 bits podemos almacenar 4 enteros)

Instrucciones ejecutadas: 11 + 1000·5 = 5.011

Instrucciones SIMD

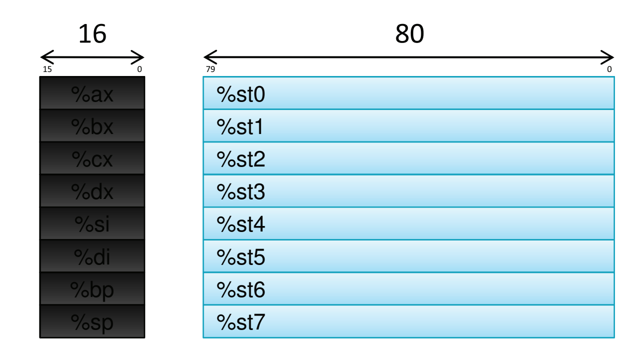
- **Extensiones del lenguaje máquina:**
  - i8086 (1977)



[redacted]

Instrucciones SIMD

- **Extensiones del lenguaje máquina:**
  - i8086 (1977), i8087 (1980)

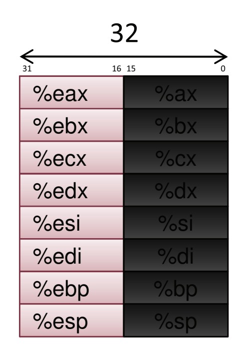


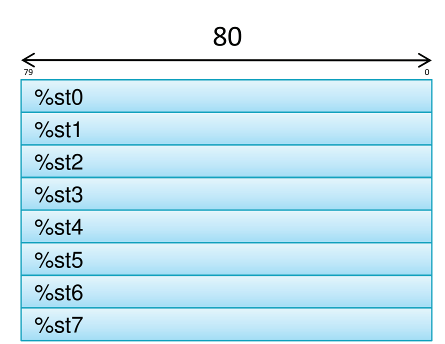
Coprocesador de coma flotante: simple precisión (32 bits) doble precisión (64 bits), precisión extendida (80 bits). Añade 60 instrucciones. Híbrido entre un banco de registros de propósito general y máquina de pila (register stack).



Instrucciones SIMD

- **Extensiones del lenguaje máquina:**
  - i8086 (1977), i8087 (1980), IA-32 (1985)



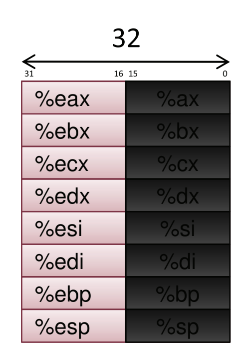


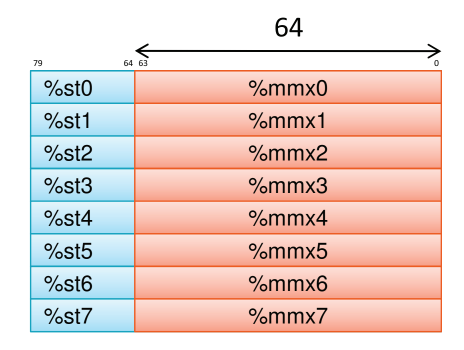
El i80386 extiende la arquitectura a 32 bits (IA-32). Se añaden modos de direccionamiento e instrucciones. La extensión IA-32 es una máquina de registros de propósito general (con alguna excepción).

Instrucciones SIMD

Extensiones del lenguaje máquina:

- i8086 (1977), i8087 (1980), IA-32 (1985), MMX (1997)





Instrucciones SIMD de 64 bits para enteros (8x8, 4x16, 2x32). Los registros %mmx están mapeados sobre los registros de punto flotante (sobre los 64 bits de mantisa)

Instrucciones SIMD

Extensiones del lenguaje máquina:

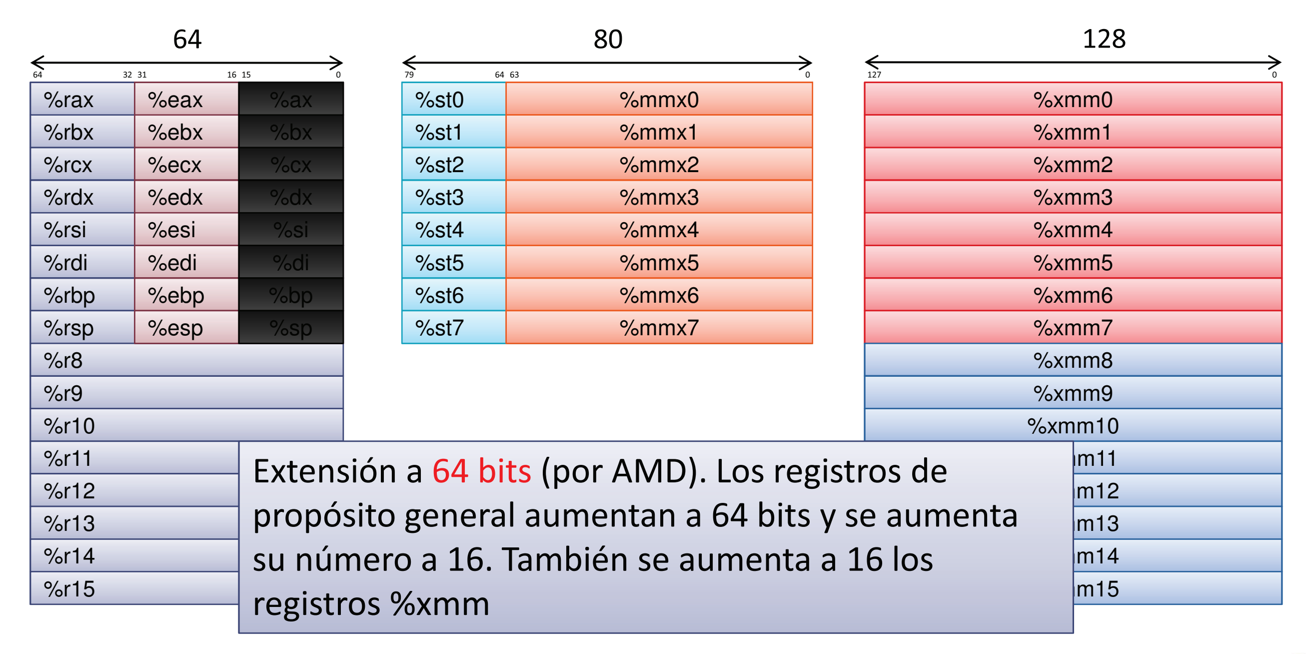
- i8086 (1977), i8087 (1980), IA-32 (1985), MMX (1997), SSE (1999)

32 (31 … 16 15 … 0)		80 (79 … 64 63 … 0)		128 (127 … 0)
%eax	%ax	%st0	%mmx0	%xmm0
%ebx	%bx	%st1	%mmx1	%xmm1
%ecx	%cx	%st2	%mmx2	%xmm2
%edx	%dx	%st3	%mmx3	%xmm3
%esi	%si	%st4	%mmx4	%xmm4
%edi	%di	%st5	%mmx5	%xmm5
%ebp	%bp	%st6	%mmx6	%xmm6
%esp	%sp	%st7	%mmx7	%xmm7

Nuevas instrucciones SIMD de 128 bits para enteros (16x8,8x16,4x32,2x64) y punto flotante (4x32,2x64). Usan un banco de registros separado. Se han ido incorporando nuevas instrucciones en distintas generaciones: SSE, SSE2, SSE3, SSSE3, SSE4 (4.1, 4.2, 4a), …….

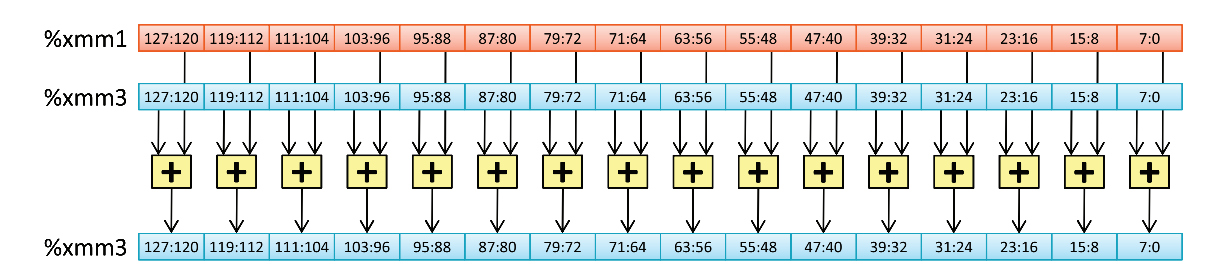
Instrucciones SIMD

- **Extensiones del lenguaje máquina:**
  - i8086 (1977), i8087 (1980), IA-32 (1985), MMX (1997), SSE (1999), AMD-64 (2003)



Instrucciones SIMD

- Ejemplo: **`paddb %xmm1, %xmm3`** # add packed byte integers



Instrucciones	Descripción	Notas
PADDB	add packed byte integers	16x8bits
PADDW	add packed word integers	8x16bits
PADDD	add packed doubleword integers	4x32bits
PADDQ	add packed quadword integers	2x64bits
ADDPS	add packed single-precission floating-point values	4x32 bits
ADDPD	add packed double-precission floating-point values	2x64bits

Instrucciones SIMD

- Ejemplo: Calcular el vector máximo de 2 vectores de caracteres

```
void maxv(char a[], b[], max[]) {
  int i;
  for (i=0;i<8000;i++) {
    if (a[i]>b[i]) max[i]=a[i];
    else max[i]=b[i];
  }
}
```

```
for:  cmpl $8000, %esi
      jge fin
      movdqa (%ebx, %esi),%xmm0
      pmaxsb (%ecx, %esi),%xmm0
      movdqa %xmm0, (%edx, %esi)
      addl $16, %esi
      jmp for
```

9 instrucciones cada iteración

vs

7 instrucciones cada 16 iteraciones

```
maxv: pushl %ebp
      movl %esp, %ebp
      ; Salvar Registros
      movl 8(%ebp),%ebx       ; ebx ← @a
      movl 12(%ebp),%ecx      ; ebx ← @b
      movl 16(%ebp),%edx      ; edx ← @max
      xorl %esi,%esi          ; i ← 0 ret

for:  cmpl $8000, %esi        ; i < 8000
      jge fin
      movb (%ebx, %esi),%al   ; al ← a[i]
      cmpb (%ecx, %esi),%al   ; a[i] > b[i]
      jle else
      movb %al, (%edx, %esi); max[i] ← a[i]
      jmp cont
else: movb (%ecx, %esi), %al;
      movb %al, (%edx, %esi); max[i] ← b[i]
cont: incl %esi               ; i++
      jmp for

fin:  ; Restaurar Registros
      popl %ebp
      ret
```


Tipos de datos estructurados

- **Estructuras (struct)**
  - conjunto heterogéneo de datos
    - ✓ almacenados de forma contigua en memoria
    - ✓ referenciados por su nombre

Ejemplo:

```
typedef struct {
  int i;
  int a[3];
  int *p;
} X;
X S;

Init(&S);
```

Tamaño	Campo	Desplazamiento
4 bytes	i	+0
12 bytes	a[0]	+4
	a[1]	
	a[2]	
4 bytes	p	+16

Ejemplo:

```
void Init (X *S) {
  (*S).i = 1;
  S->a[2] = 0;
  S->p = &(*S).a[0];
}
```

Tipos de datos estructurados

- **Estructuras (struct)**

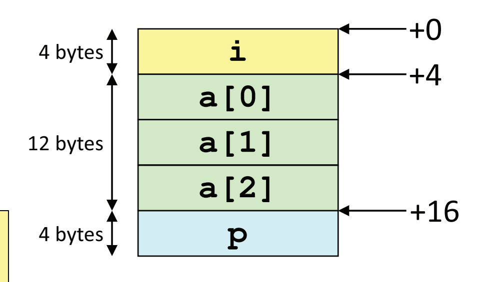
Ejemplo:

```
typedef struct {
   int i;
   int a[3];
   int *p;
} X;
X S;

Init(&S);
```

Traducción:

```
Init: push  %ebp
      movl  %esp,%ebp
      movl  8(%ebp),%edx
      movl  $1,(%edx)
      movl  $0,12(%edx)
      leal  4(%edx),%eax
      movl  %eax,16(%edx)
      popl  %ebp
      ret
```



Ejemplo:

```
void Init (X *S) {
   (*S).i = 1;
   S->a[2] = 0;
   S->p = &(*S).a[0];
}
```


Alineamiento de datos

- **Alineamiento de datos**
  - Un tipo de dato primitivo requiere k bytes
    - ✓ La dirección debe ser múltiplo de k
    - ✓ En algunas máquinas es obligatorio. Aconsejable en IA32
    - ✓ Trato distinto en Windows y Linux
- **Motivación para alinear datos**
  - Accesos a memoria por longword o quadwords alineados
  - Accesos no alineados pueden provocar que un mismo dato se encuentre en 2 líneas de cache diferentes.
  - Memoria virtual: problemas si el dato está entre dos páginas
- **Compilador**
  - Inserta “espacios” en la estructura para asegurar que los datos están alineados.

Alineamiento de datos

- **Alineamiento en linux-32 (gcc):**
  - char (1 byte): alineado a 1-byte (no hay restricciones en la @)
  - short (2 bytes): alineado a 2-bytes (el bit más bajo de la @ debe ser 0
  - int (4 bytes): alineado a 4-bytes (los 2 bits más bajos de la @ deben ser 00)
  - puntero (4 bytes): alineado a 4-bytes
  - double (8 bytes): alineado a 4-bytes
  - long double (12 bytes): alineado a 4-bytes
- **Diferencias linux-64:**
  - double (8 bytes): alineado a 8-bytes.
  - long double (16 bytes): alineado a 16-bytes.
  - puntero (8 bytes): alineado a 8-bytes.
- **Diferencias windows-32:**
  - double (8 bytes): alineado a 8-bytes.
  - long double (10 bytes): alineado a 2-bytes

Alineamiento de datos

- **Offsets dentro de una estructura:**
  - deben satisfacer los requerimientos de alineamiento de sus elementos
- **Dirección de la estructura**
  - Cada estructura tiene un requerimiento de alineamiento k (el mayor de los alineamientos de cualquier elemento)
  - La @ inicial de la estructura debe ser múltiplo de k

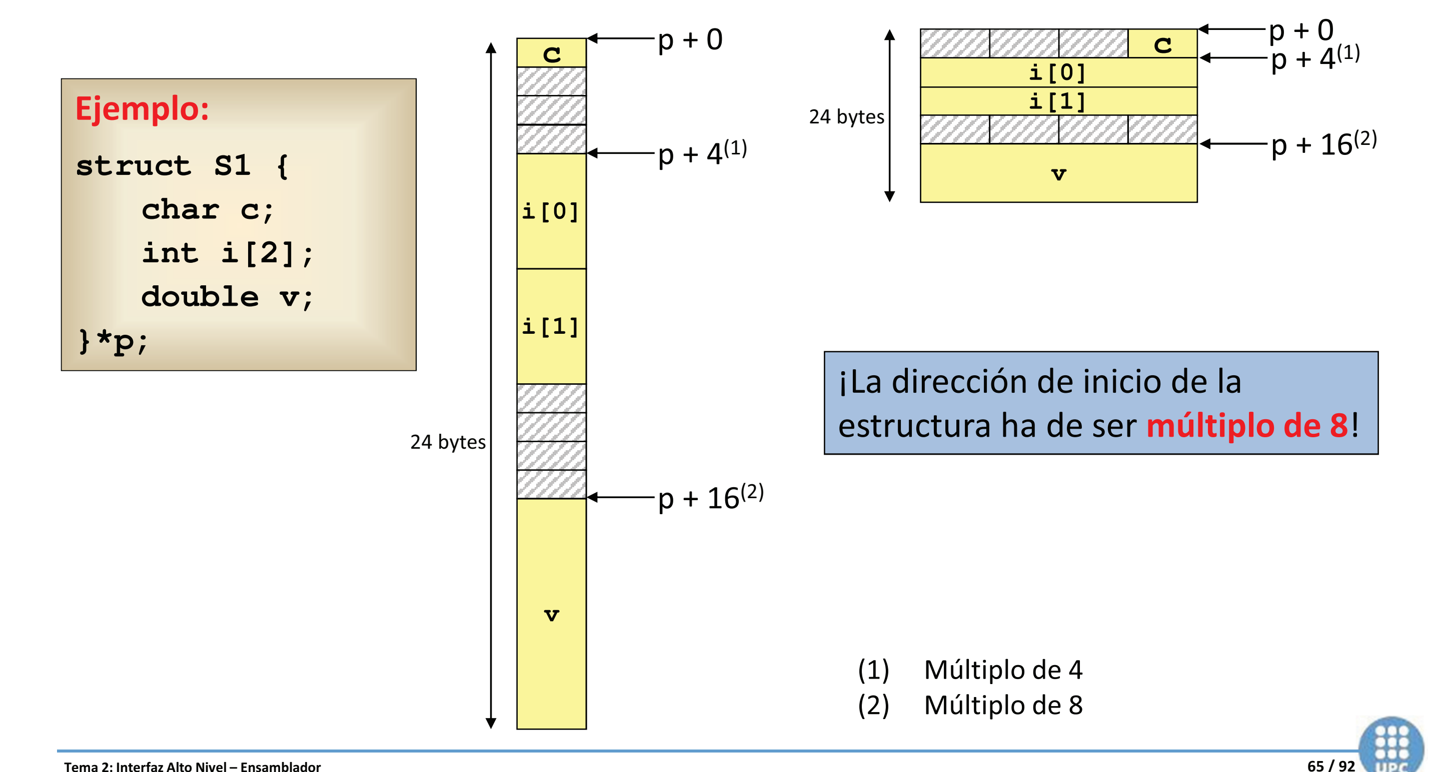
Ejemplo:

```
struct S1 {
    char c;
    int i[2];
    double v;
}*p;
```

- (Linux-32) k = 4
- (Linux-64) k = 8 debido al elemento *double*

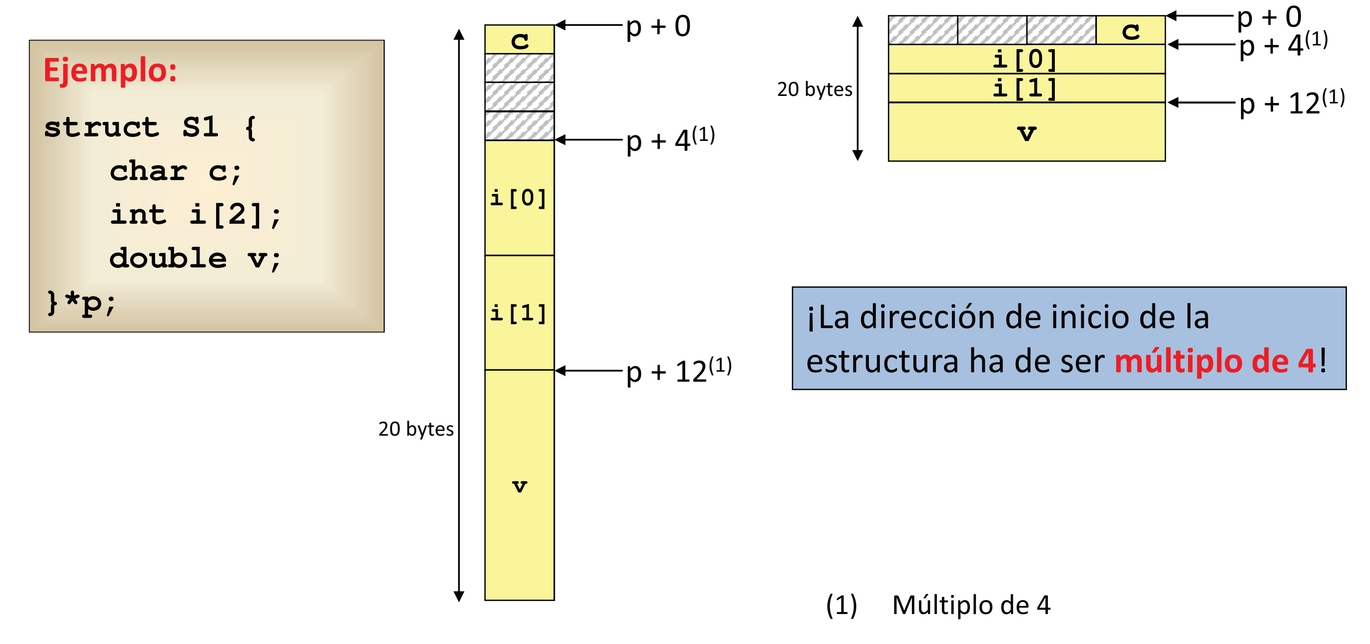
Alineamiento de datos

- **Ejemplo (Linux-64) k = 8 debido al elemento *double***



Alineamiento de datos

- **Ejemplo (Linux-32) k = 4 debido a que el elemento *double* se trata a nivel de alineamiento como un elemento de 4 bytes.**

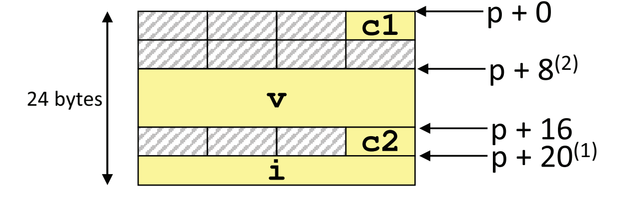


Alineamiento de datos

- **El orden de los elementos de una estructura influye en su tamaño.**
  - Ejemplo en Linux-64

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```



16 bytes
v
c2 c1
i
p + 0
p + 8
p + 12(1)

(1) Múltiplo de 4
(2) Múltiplo de 8

¡La dirección de inicio de la estructura ha de ser **múltiplo de 8**!

Alineamiento de datos

- **El orden de los elementos de una estructura influye en su tamaño.**


```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

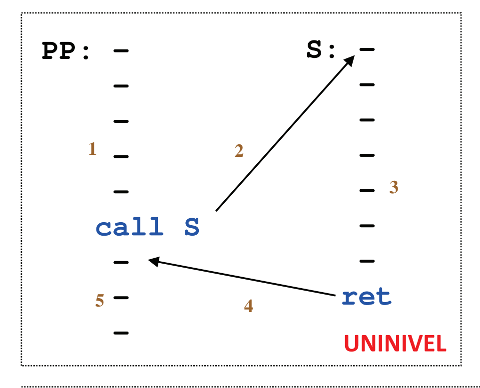

- El programador de C puede reordenar los elementos de la estructura para minimizar el espacio ocupado.
- Sin embargo, el programador de ensamblador **NO** puede realizar esta optimización cuando enlaza ensamblador con C.

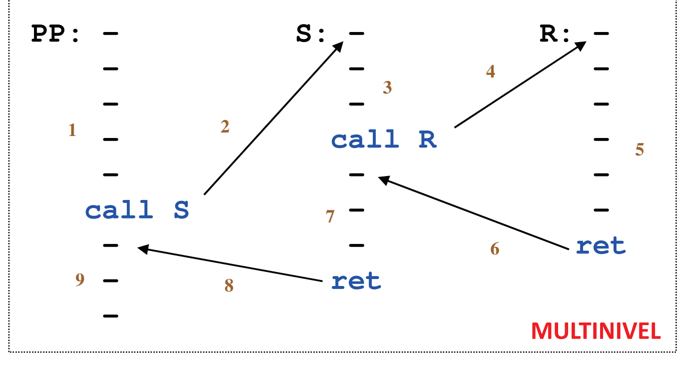
Gestión de subrutinas

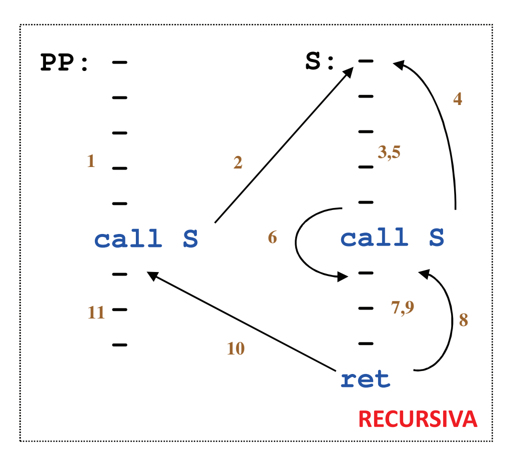
- **Subrutina: Conjunto de instrucciones de LM que realiza una tarea específica y que puede ser activada (llamada) desde cualquier punto de un programa o desde la propia subrutina**
- **Activación interna: la llamada se hace desde la propia subrutina**
- **Activación externa: la llamada se hace desde el programa principal o desde otra subrutina**
- **Entre el 5 y el 10% de las instrucciones que ejecuta un procesador son llamadas o retornos de subrutinas.**
- **Clasificación de las subrutinas**
  - Uninivel
  - Multinivel
  - Recursivas
  - Reentrantes
  - No reentrantes

Gestión de subrutinas

- **Clasificación de las subrutinas**







Gestión de subrutinas

Ventajas del uso de subrutinas

- El código ocupa menos espacio en memoria
- El código está más estructurado
  - ✓ facilidad de depuración
  - ✓ facilidad de expansión o modificación
  - ✓ posibilidad de usar librerías públicas
- El LM refleja la idea fundamental de los lenguajes estructurados de alto nivel: la existencia de funciones y procedimientos

Inconvenientes del uso de subrutinas

- El tiempo de ejecución de los programas aumenta debido a:
  - ✓ la ejecución de las instrucciones de llamada y retorno de subrutina
  - ✓ el paso de parámetros
- La complejidad del procesador es mayor porque debe añadirse hardware específico para la gestión eficiente de subrutinas

Gestión de subrutinas

Terminología

- Parámetros
  - ✓ Valor
  - ✓ Referencia
- Variables locales
- Invocación
- Retorno resultado
- Cuerpo subrutina

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;
  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];
  return sum;
}

void PDOT(int M[10][10], int *p) {
  int i;
  *p = 0;
  for (i=0; i<10; i++)
    *p += DOT(&M[0][0],&M[i][0],10);
}
```

Gestión de subrutinas

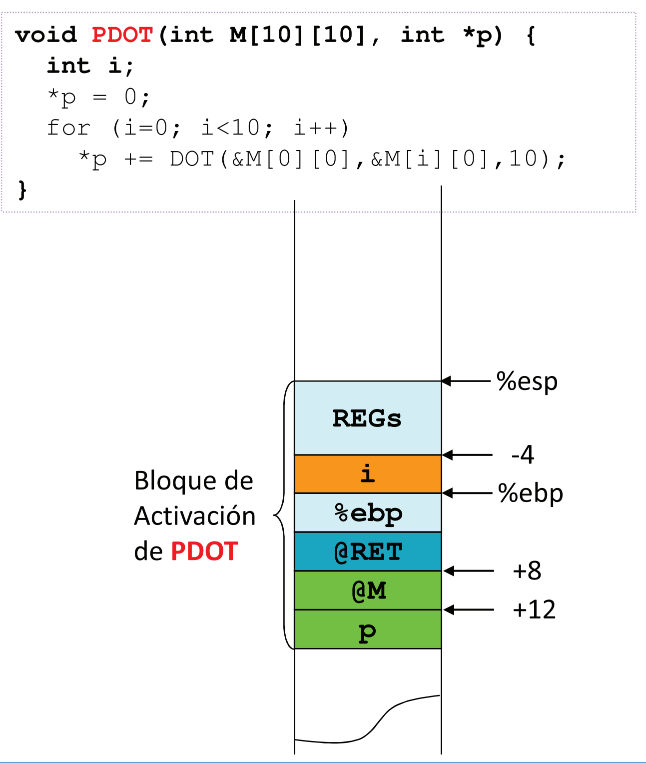
- **Convenciones en C-Linux 32 bits**
  - Los parámetros se pasan por la pila de derecha a izquierda
  - Los vectores y matrices siempre se pasan por referencia
  - Los structs se pueden pasar por valor, no importa el tamaño
  - Los parámetros de tipo caracter (1 byte) ocupan 4 bytes
  - Los parámetros de tipo short (2 bytes) ocupan 4 bytes
  - Las variables locales están alineadas en la pila con la misma convención que dentro de un struc
    - ✓ Char en cualquier dirección
    - ✓ Short en direcciones múltiplos de 2
    - ✓ Integer en direcciones múltiplos de 4
  - Los registros %ebp, %esp se salvan siempre implícitamente en la gestión de subrutinas
  - Los registros %ebx, %esi, %edi se han de salvar si son modificados
  - Los registros %eax, %ecx, %edx se pueden modificar en el interior de una subrutina. Si es necesario, el LLAMADOR ha de salvarlos
  - Los resultados se devuelven siempre en %eax

Gestión de Subrutinas

- **Bloque activación**
  - PILA

```
         {código llamador PDOT}
         empilar parámetros PDOT
         call PDOT
         ...

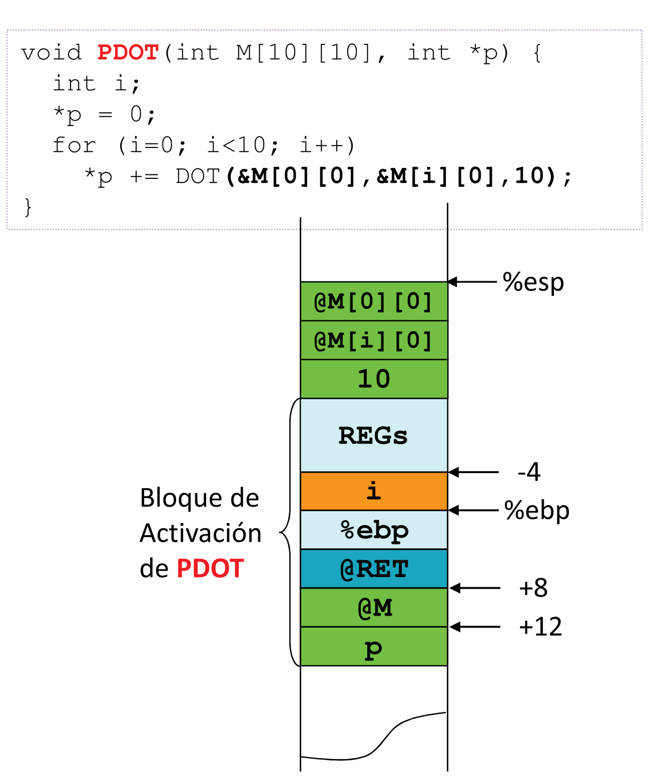
PDOT:    pushl %ebp
         movl %esp, %ebp
         subl $4, %esp
         salvar registros
         –
         –
         –
```



Gestión de Subrutinas

1. Paso de parámetros

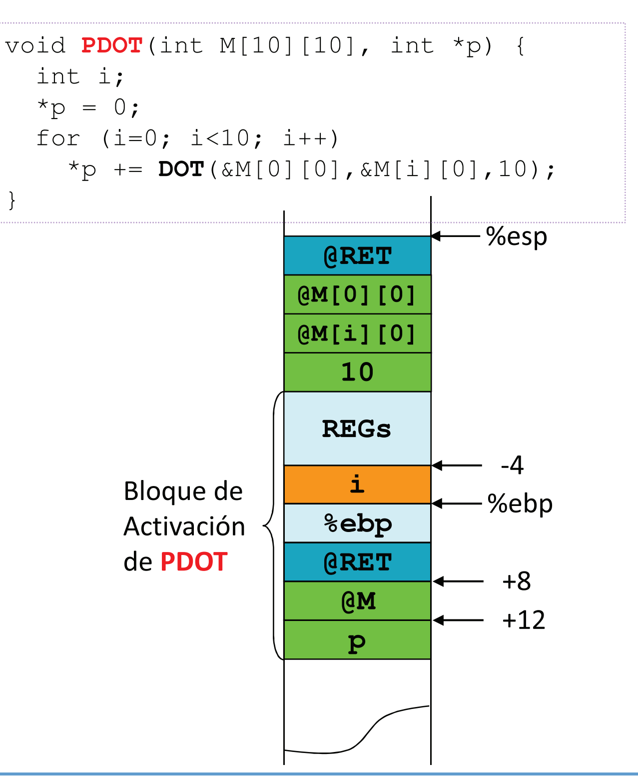
```
PDOT:  –
       –
       –
       pushl $10
       imull $10,-4(%ebp),%edx
       movl 8(%ebp),%ebx
       leal (%ebx,%edx,4),%eax
       pushl %eax
       pushl %ebx
```



Gestión de Subrutinas

2. Llamada a la subrutina

```
PDOT:  –
       –
       –
       pushl $10
       imull $10,-4(%ebp),%edx
       movl 8(%ebp),%ebx
       leal (%ebx,%edx,4),%eax
       pushl %eax
       pushl %ebx
       call DOT
```



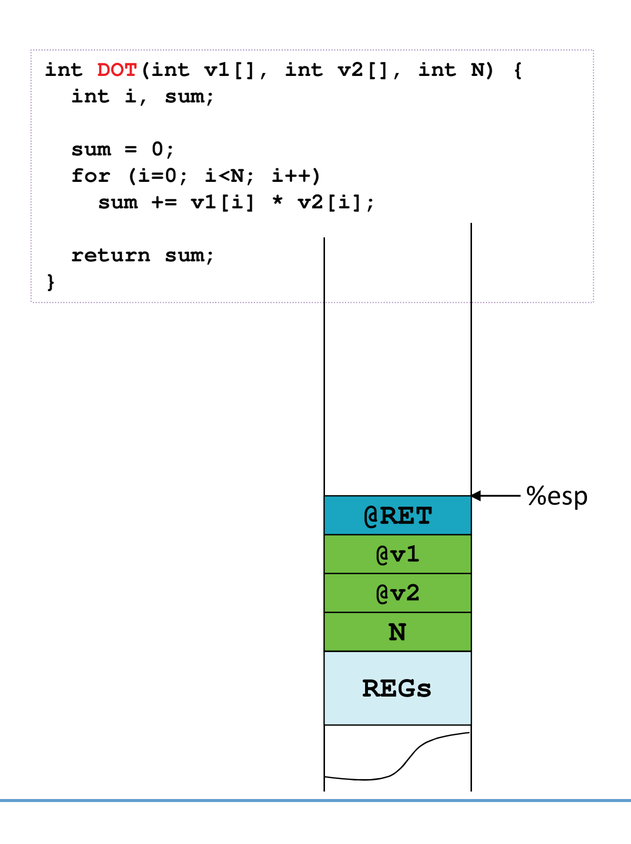
2. Saltamos a la subrutina

```
DOT:
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



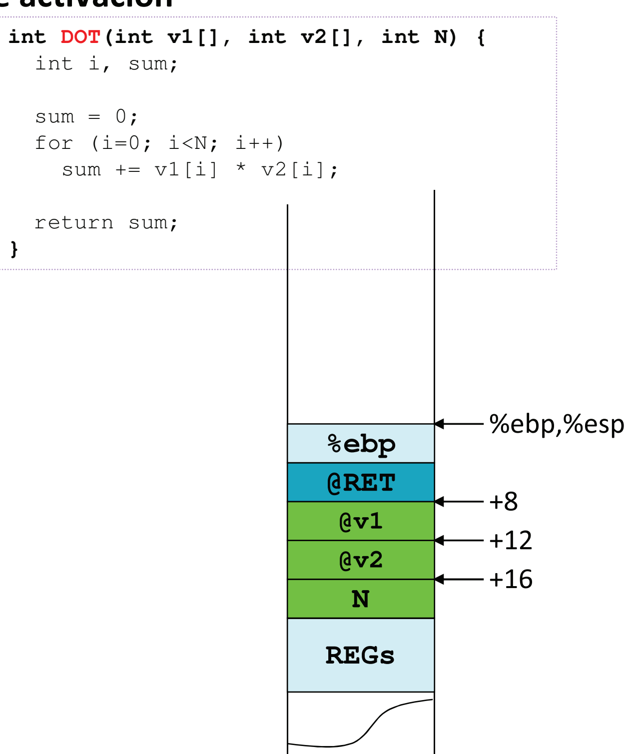
3. Enlace dinámico y puntero al bloque de activación

```
DOT: pushl %ebp
     movl %esp, %ebp
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

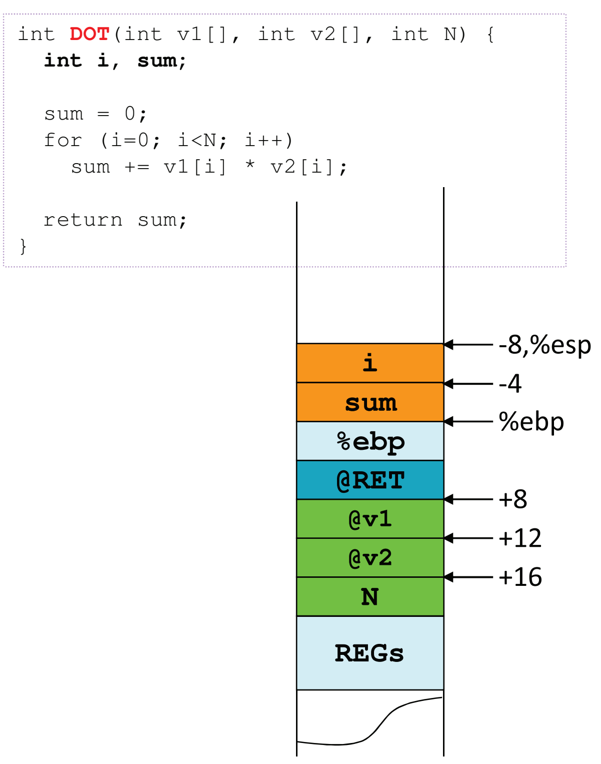
  return sum;
}
```



Gestión de Subrutinas

4. Reserva espacio para variables locales

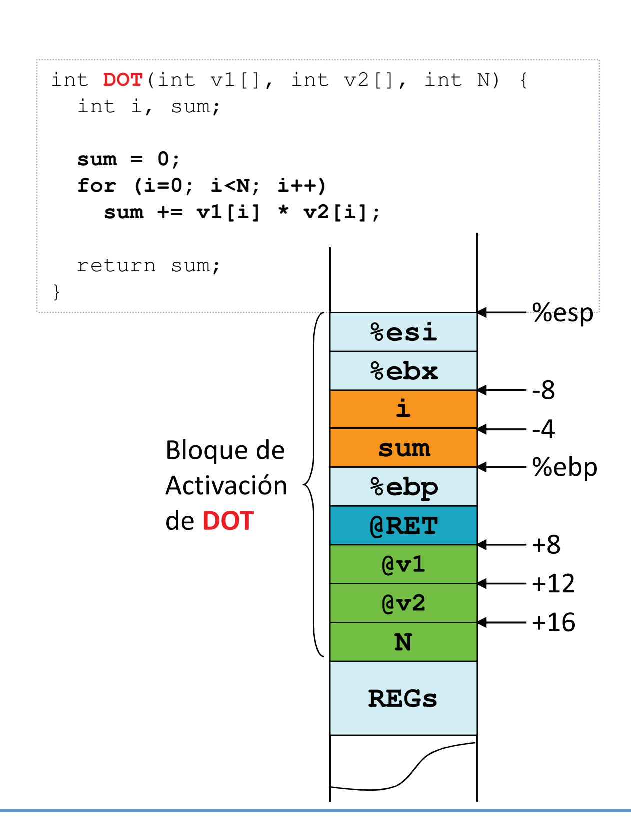
```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
```



Gestión de Subrutinas

5. Salvar estado del llamador

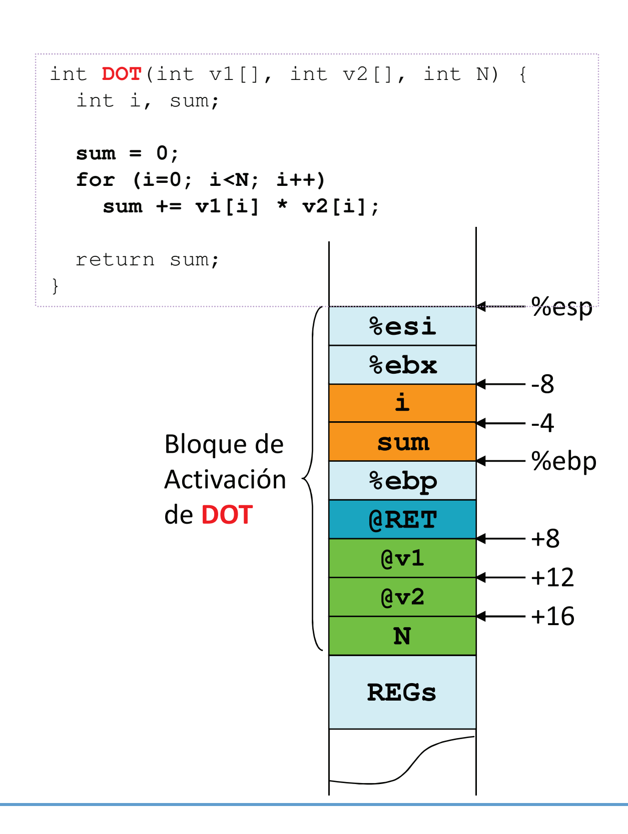
```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
```



Gestión de Subrutinas

6. Cuerpo subrutina

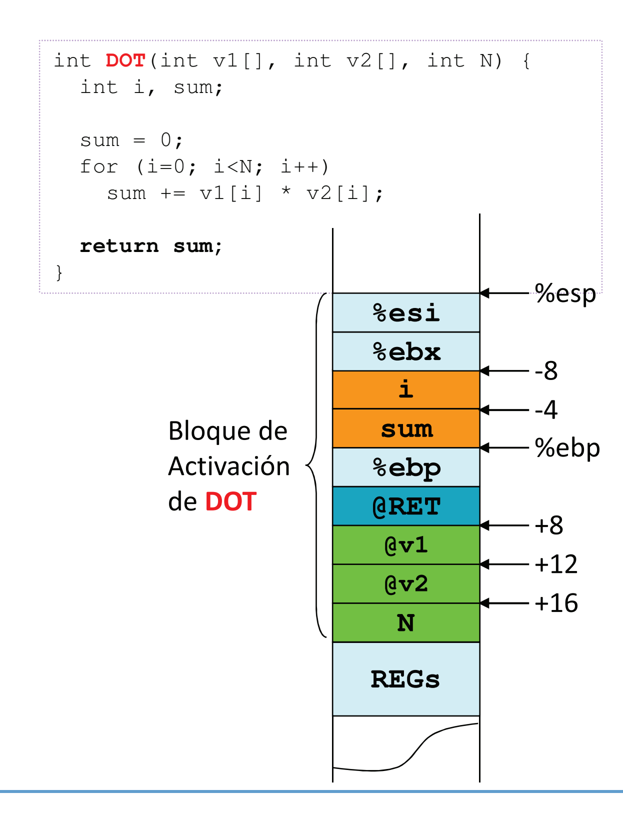
```
DOT:  pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp),%ebx
      movl 12(%ebp),%esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp),%edx
      jge end
      movl (%esi,%edx,4),%ecx
      imull (%ebx,%edx,4),%ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end:
```



Gestión de Subrutinas

7. Mover resultado a %eax

```
DOT:  pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp),%ebx
      movl 12(%ebp),%esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp),%edx
      jge end
      movl (%esi,%edx,4),%ecx
      imull (%ebx,%edx,4),%ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end:  movl -4(%ebp),%eax
```



Gestión de Subrutinas

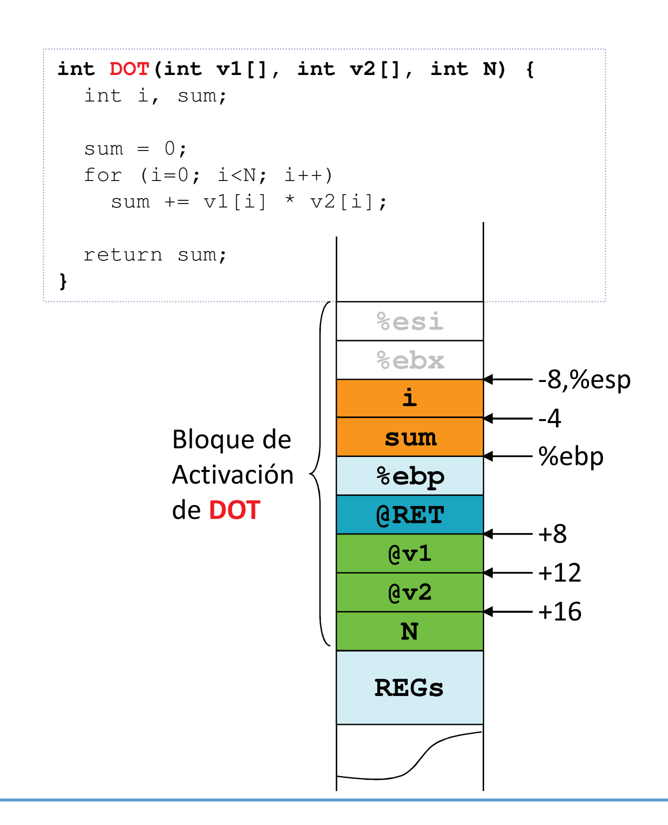
8. Restaurar estado llamador

```
DOT:  pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp),%ebx
      movl 12(%ebp),%esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp),%edx
      jge end
      movl (%esi,%edx,4),%ecx
      imull (%ebx,%edx,4),%ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end:  movl -4(%ebp),%eax
      popl %esi
      popl %ebx
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



Gestión de Subrutinas

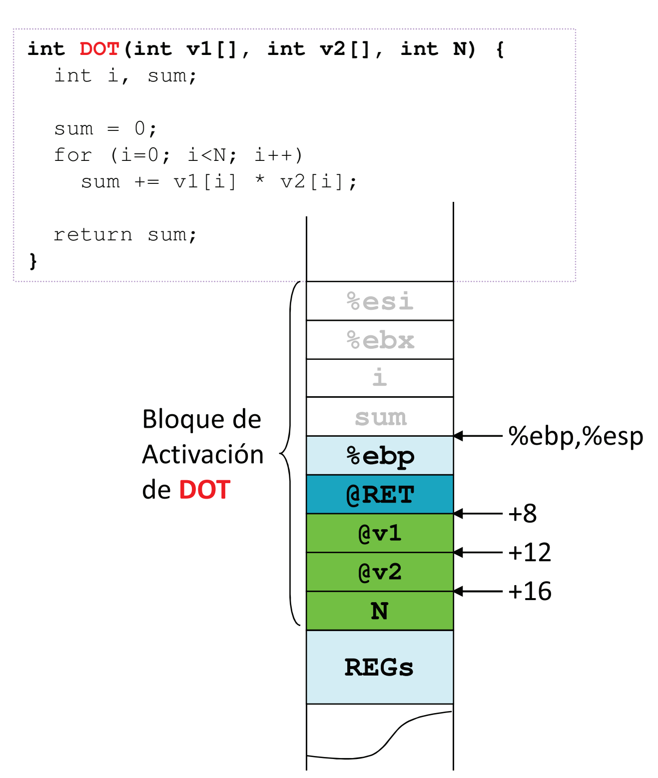
9. Eliminar variables locales

```
DOT:  pushl %ebp
      movl %esp, %ebp
      subl $8, %esp
      pushl %ebx
      pushl %esi
      movl 8(%ebp),%ebx
      movl 12(%ebp),%esi
      movl $0,-4(%ebp)
      xorl %edx,%edx
for:  cmpl 16(%ebp),%edx
      jge end
      movl (%esi,%edx,4),%ecx
      imull (%ebx,%edx,4),%ecx
      addl %ecx,-4(%ebp)
      incl %edx
      jmp for
end:  movl -4(%ebp),%eax
      popl %esi
      popl %ebx
      movl %ebp,%esp
```

```
int DOT(int v1[], int v2[], int N) {
  int i, sum;

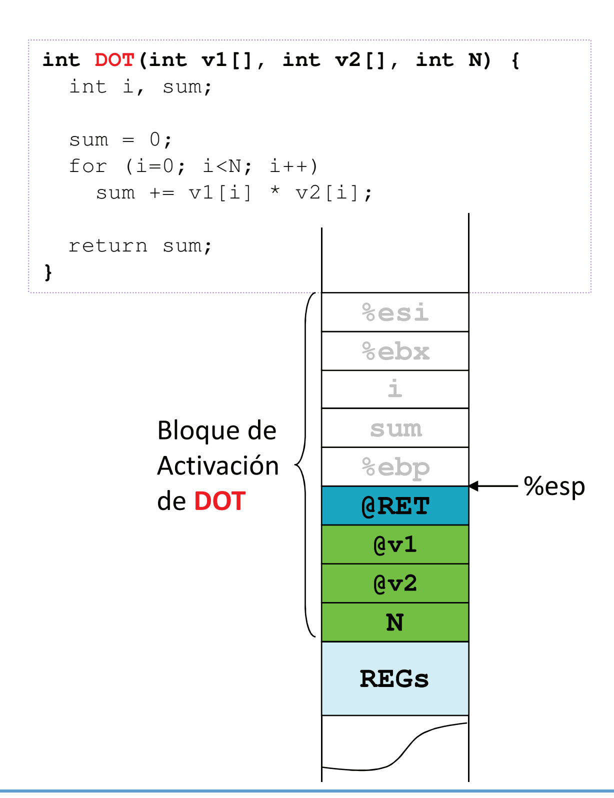
  sum = 0;
  for (i=0; i<N; i++)
    sum += v1[i] * v2[i];

  return sum;
}
```



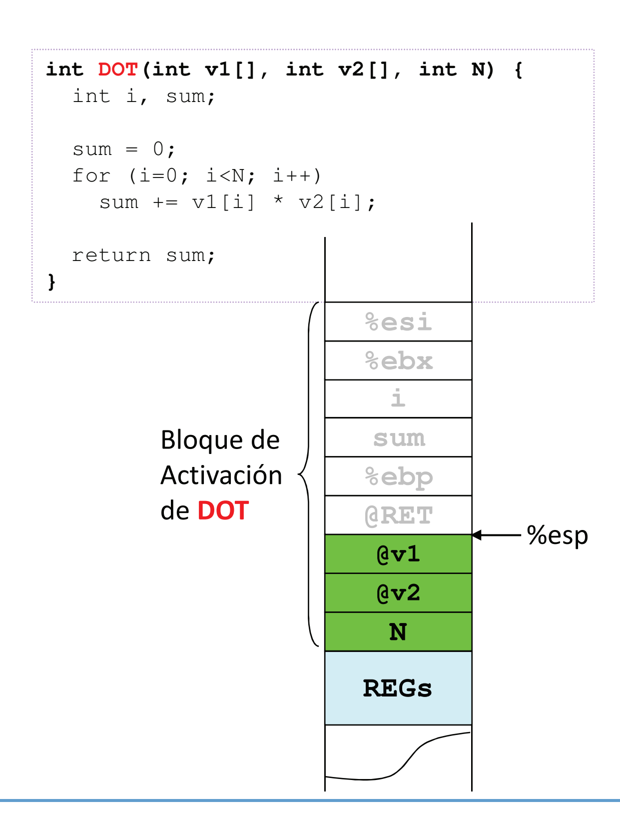
10. Deshacer enlace dinámico

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
     popl %esi
     popl %ebx
     movl %ebp,%esp
     popl %ebp
```



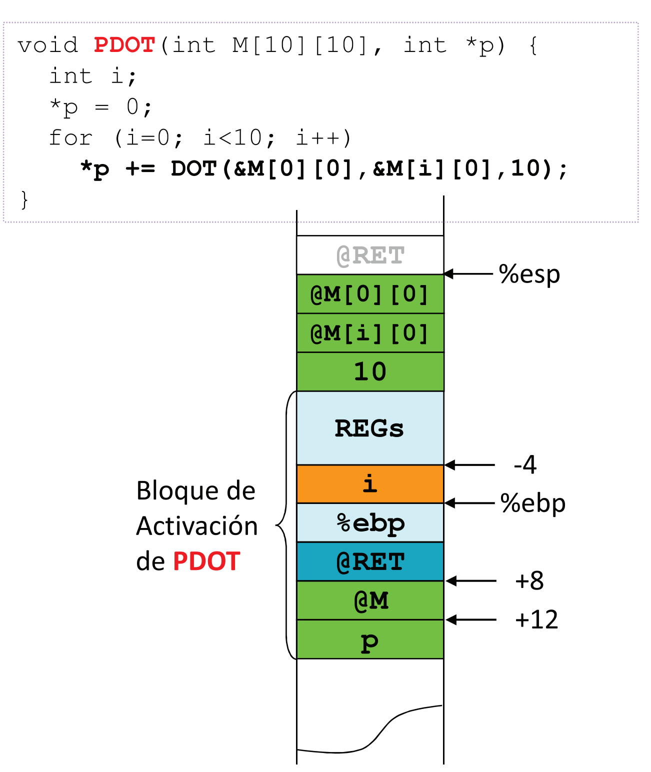
11. Retorno subrutina

```
DOT: pushl %ebp
     movl %esp, %ebp
     subl $8, %esp
     pushl %ebx
     pushl %esi
     movl 8(%ebp),%ebx
     movl 12(%ebp),%esi
     movl $0,-4(%ebp)
     xorl %edx,%edx
for: cmpl 16(%ebp),%edx
     jge end
     movl (%esi,%edx,4),%ecx
     imull (%ebx,%edx,4),%ecx
     addl %ecx,-4(%ebp)
     incl %edx
     jmp for
end: movl -4(%ebp),%eax
     popl %esi
     popl %ebx
     movl %ebp,%esp
     popl %ebp
     ret
```



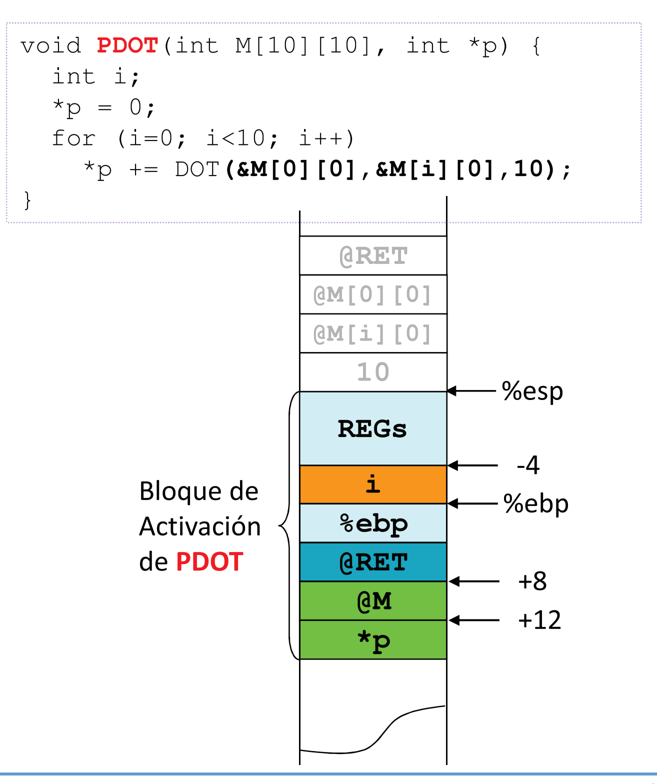
11. Volvemos a la subrutina

```
PDOT:  –
       –
       –
       pushl $10
       imull $10,-4(%ebp),%edx
       movl 8(%ebp),%ebx
       leal (%ebx,%edx,4),%eax
       pushl %eax
       pushl %ebx
       call DOT
```



12. Eliminar parámetros

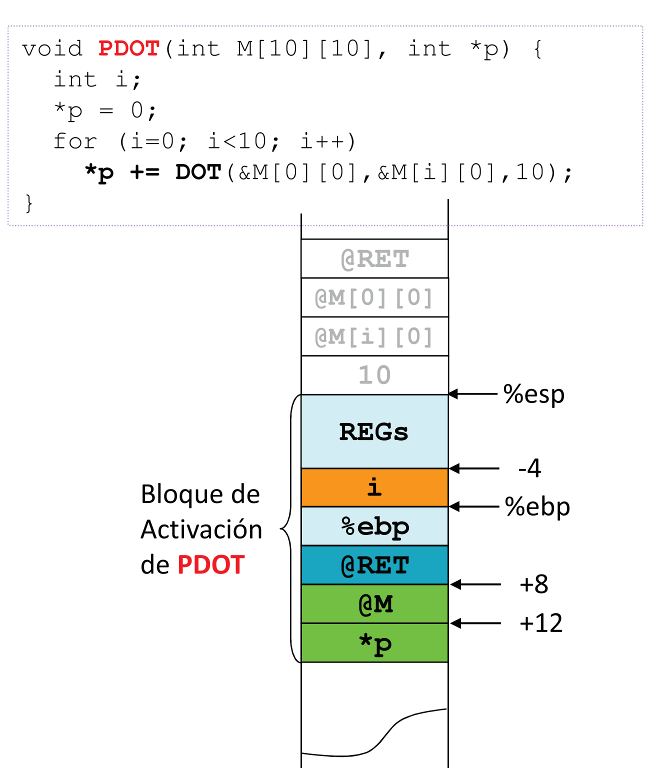
```
PDOT:  –
       –
       –
       pushl $10
       imull $10,-4(%ebp),%edx
       movl 8(%ebp),%ebx
       leal (%ebx,%edx,4),%eax
       pushl %eax
       pushl %ebx
       call DOT
       addl $12,%esp
```



Gestión de Subrutinas

13. Recoger/usar resultado

```
PDOT:  –
       –
       –
       pushl $10
       imull $10,-4(%ebp),%edx
       movl 8(%ebp),%ebx
       leal (%ebx,%edx,4),%eax
       pushl %eax
       pushl %ebx
       call DOT
       addl $12,%esp
       movl 12(%ebp),%ebx
       addl %eax,(%ebx)
```



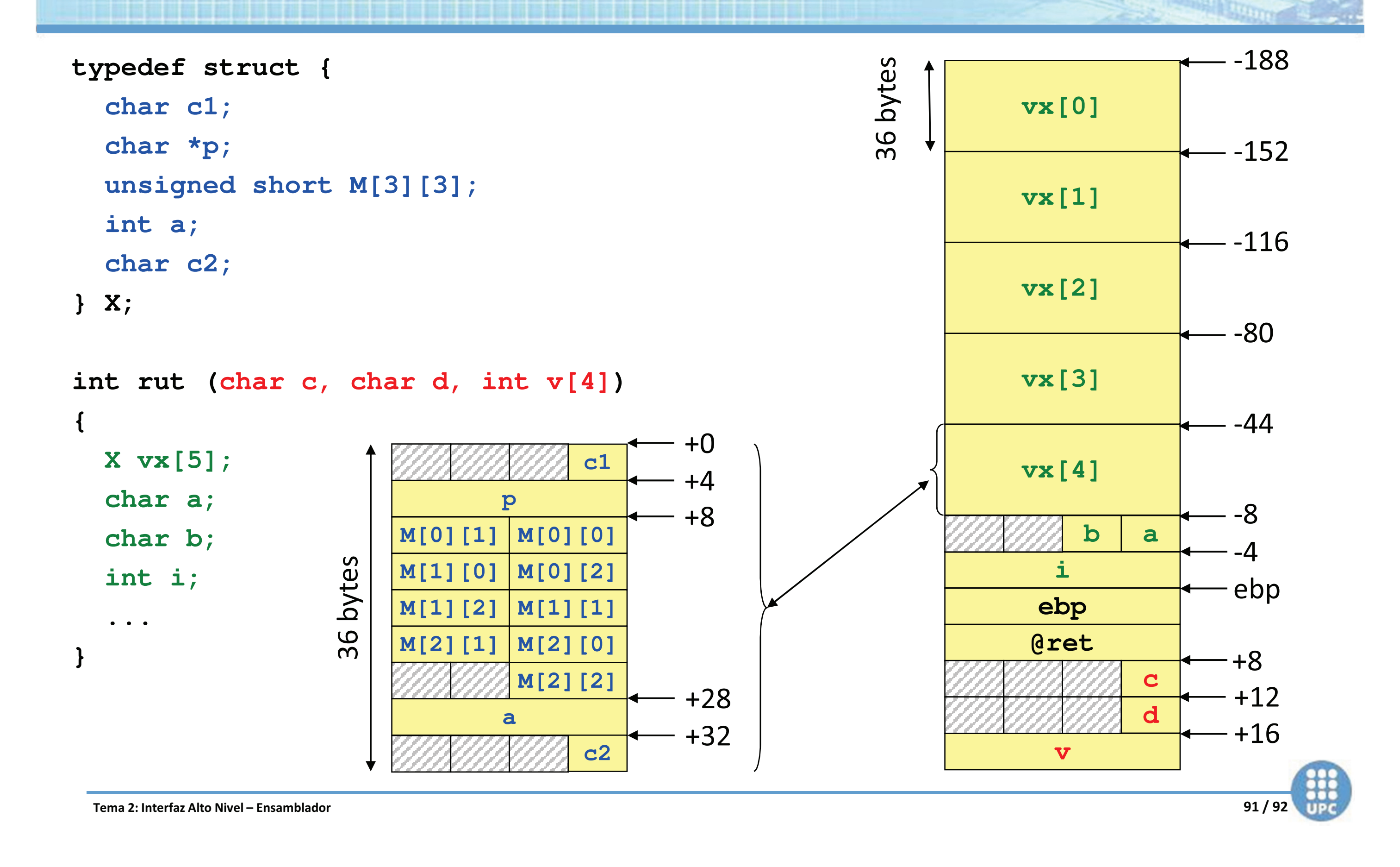
Gestión de subrutinas

PDOT:

- -
- -
- -
- 1 Paso de parámetros
- 2 llamada subrutina
- 12 elimina parámetros
- 13 Recoger/usar resultado
- -
- -

DOT:

- 3 Enlace dinámico, puntero bloque de activación
- 4 Reserva espacio variables locales
- 5 Salvar estado llamador
- 6 Cuerpo subrutina
- 7 Mover resultado a eax
- 8 Restaura estado
- 9 elimina variables locales
- 10 Deshacer enlace dinámico
- 11 retorno de subrutina



Gestión de Subrutinas

```
typedef struct {
  char c1;
  char *p;
  unsigned short M[3][3];
  int a;
  char c2;
} X;

int rut2 (X sx, X *px)
{
...
}
```

ebp			← ebp
@ret			
		c1	← +8
p			← +12
M[0][1]		M[0][0]	← +16
M[1][0]		M[0][2]	
M[1][2]		M[1][1]	
M[2][1]		M[2][0]	
		M[2][2]	
a			← +36
		c2	← +40
px			← +44

sx = { c1 … c2 }